# Interprocedural Optimization in GCC

Honza Hubička

SuSE ČR s.r.o
Prague

Gelato ICE, 2007

# Outline

# Interprocedural Optimization

# Interprocedural Optimization

Interprocedural analysis (IPA) and optimization is about optimizing across function boundaries.

# Interprocedural Optimization

Interprocedural analysis (IPA) and optimization is about optimizing across function boundaries.
Basic optimizations include:

- Removal of unused functions and variables
- Alias analysis
- Devirtualization
- Inlining
- Constant propagation
- Register allocation
- etc. . .

# Interprocedural Optimization

Interprocedural analysis (IPA) and optimization is about optimizing across function boundaries.
Basic optimizations include:

- Removal of unused functions and variables
- Alias analysis
- Devirtualization
- Inlining
- Constant propagation
- Register allocation
- etc. . .

In general simple but effective transformations cheap enough to be performed on large amount of code.

# Different scopes of IPA

- **Function at a time** — functions are compiled as early as are available, allowing only forward propagation and inlining.

# Different scopes of IPA

- **Function at a time** — functions are compiled as early as are available, allowing only forward propagation and inlining.
- **Unit at a time** — whole compilation unit (source file or multiple sources) is parsed first, then interprocedural optimization is run.

# Different scopes of IPA

- **Function at a time** — functions are compiled as early as are available, allowing only forward propagation and inlining.
- **Unit at a time** — whole compilation unit (source file or multiple sources) is parsed first, then interprocedural optimization is run.
- **Linktime** — object files are replaced by “fake” objects containing intermediate language. Linker then calls back to compiler to optimize all object files together. Allows i.e. cross module inlining.

# Different scopes of IPA

- **Function at a time** — functions are compiled as early as are available, allowing only forward propagation and inlining.
- **Unit at a time** — whole compilation unit (source file or multiple sources) is parsed first, then interprocedural optimization is run.
- **Linktime** — object files are replaced by “fake” objects containing intermediate language. Linker then calls back to compiler to optimize all object files together. Allows i.e. cross module inlining.
- **Whole program** — implemented same as Linktime, compiler can however assume that it sees whole program, thus making effectively global variables and functions static. System libraries needs to be dealt with.

# GCC Way to Interprocedural optimization

GCC was designed to make IPA dificult:

- Function at a time (GCC 2.95)
- Inliner on tree level (GCC 3.0)
- Backward inlining (GCC 3.2)
- Callgraph module and rewrite of inliner heuristic
- Unit at a time (GCC 3.4)
- Intermodule optimization for C (GCC 3.4)
- Tree-SSA (GCC 4.0)
- Rewrite of inliner to operate on CFG (GCC 4.1)
- Basic IPA passes, function cloning (GCC 4.0-4.3)
- IPA-SSA (GCC 4.3)
- Gimple memory usage optimizations (GCC 4.3)
- Early optimizations (GCC 4.3)

# GCC Way to Interprocedural optimization

GCC was designed to make IPA dificult:

- Function at a time (GCC 2.95)
- Inliner on tree level (GCC 3.0)
- Backward inlining (GCC 3.2)
- Callgraph module and rewrite of inliner heuristic
- Unit at a time (GCC 3.4)
- Intermodule optimization for C (GCC 3.4)
- Tree-SSA (GCC 4.0)
- Rewrite of inliner to operate on CFG (GCC 4.1)
- Basic IPA passes, function cloning (GCC 4.0-4.3)
- IPA-SSA (GCC 4.3)
- Gimple memory usage optimizations (GCC 4.3)
- Early optimizations (GCC 4.3)

In many cases we are still about decade behind.

# GCC Way to Interprocedural optimization

Many projects are ongoing

- `lto-branch` (Link time optimization branch)
- `ipa-branch` (IP optimization infrastructure)
- `gimple-tuples-branch` (Reorganization of GIMPLE memory representation)
- `yara-branch` (New register allocator with IPA)
- Intermodule optimization on C++ (status unknown)

# GCC Way to Interprocedural optimization

Many projects are ongoing

- `lto-branch` (Link time optimization branch)
- `ipa-branch` (IP optimization infrastructure)
- `gimple-tuples-branch` (Reorganization of GIMPLE memory representation)
- `yara-branch` (New register allocator with IPA)
- Intermodule optimization on C++ (status unknown)

Other important not really started yet

- More relaxed type restrictions for C/Fortran intermodule
- Type system representing C/C++/fortran/Java/Ada at once
- Removal of front-end dependencies from middle-end (langhooks)
- Per statement granuality of some flags (-ffast-math)

# Pass queue

**During parsing**:

# Pass queue

**During parsing**:

Frontend
Callgraph module

**At end of each source unit**:

Lowering (CFG)
Unreachable functions/vars removal

# Pass queue

**During parsing**:

Frontend

Callgraph module

**At end of each source unit**:

Lowering (CFG)

Unreachable functions/vars removal

**At end of compilation unit**:

SSA conversion

Early inlining

Early optimizations

# Pass queue

**During parsing**:

Frontend

Callgraph module

**At end of each source unit**:

Lowering (CFG)

Unreachable functions/vars removal

**At end of compilation unit**:

SSA conversion

Early inlining

Early optimizations

IPA passes

# Pass queue

**During parsing**:

Frontend

Callgraph module

**At end of each source unit**:

Lowering (CFG)

Unreachable functions/vars removal

**At end of compilation unit**:

SSA conversion

Early inlining

Early optimizations

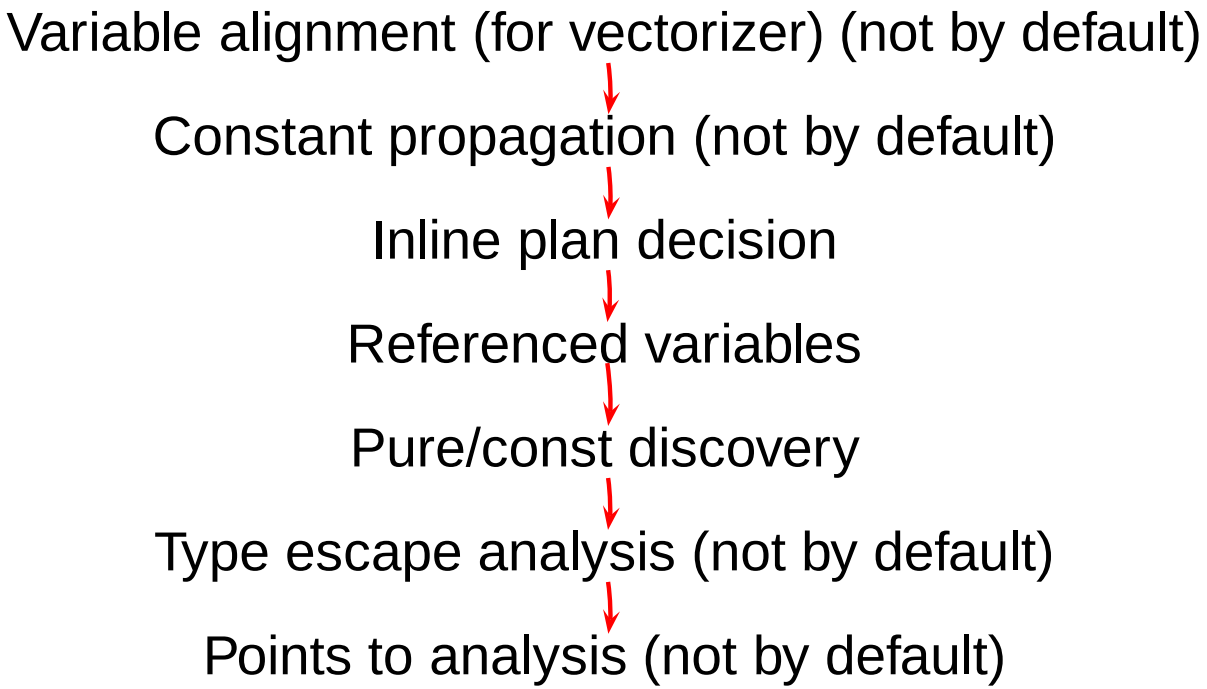
IPA passes

Inlining

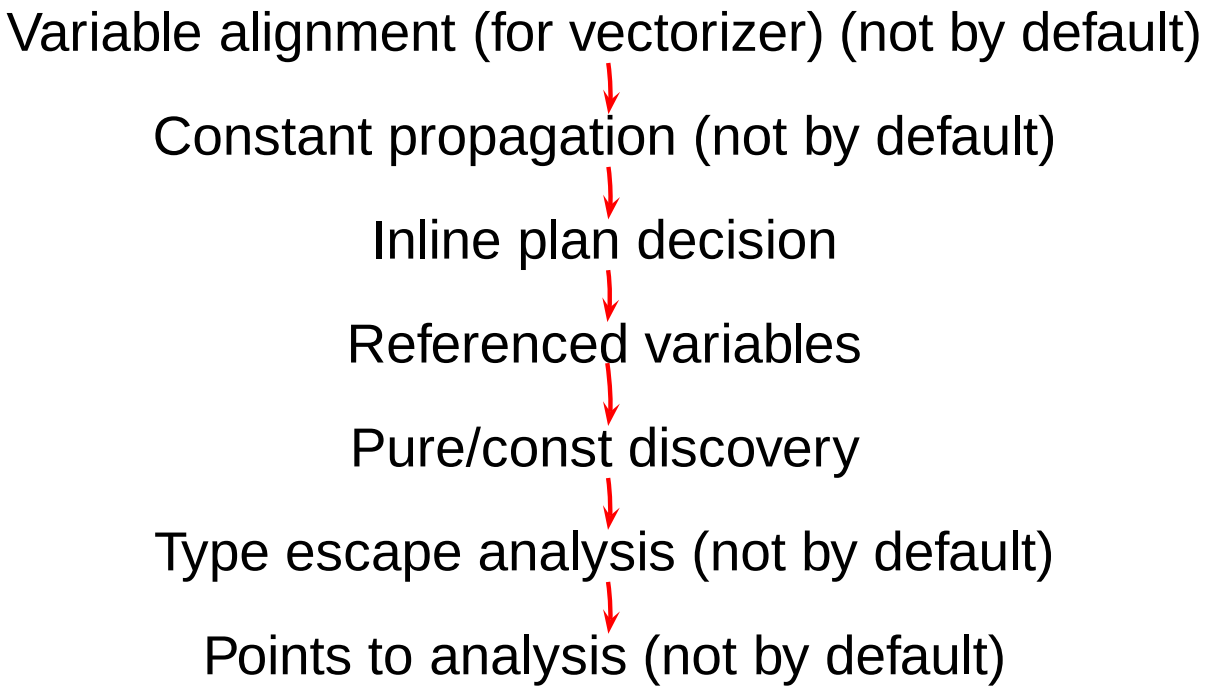
High level (GIMPLE) optimizations

Low level (RTL) optimizations

# IPA passes


Variable alignment (for vectorizer) (not by default)
Constant propagation (not by default)
Inline plan decision
Referenced variables
Pure/const discovery
Type escape analysis (not by default)
Points to analysis (not by default)

# IPA passes




Additionally some information is backward propagated across compilation done in topological order. Local functions are using non-standard calling conventions on some targets.

# Basic interface to IPA pass

As close as possible to local optimization pass

- Functions seen in SSA GIMPLE with CFG and profile

# Basic interface to IPA pass

As close as possible to local optimization pass

- Functions seen in SSA GIMPLE with CFG and profile
- Pass manager execute top level passes as IPA, subpasses are executed for each function (as local passes)
- Local TODO flags executed for each pass

# Basic interface to IPA pass

As close as possible to local optimization pass

- Functions seen in SSA GIMPLE with CFG and profile
- Pass manager execute top level passes as IPA, subpasses are executed for each function (as local passes)
- Local TODO flags executed for each pass
- Plan to slowly migrate API to take `struct function` as argument. For now use `push_cfun` and `current_function_decl`.
- Plan to make local alias analysis available if memory allows

# Basic interface to IPA pass

As close as possible to local optimization pass

- Functions seen in SSA GIMPLE with CFG and profile
- Pass manager execute top level passes as IPA, subpasses are executed for each function (as local passes)
- Local TODO flags executed for each pass
- Plan to slowly migrate API to take `struct function` as argument. For now use `push_cfun` and `current_function_decl`.
- Plan to make local alias analysis available if memory allows

Expose more stress on local dataflow datastructure memory use.

# Basic interface to IPA pass

IPA interface

- Callgraph
  - Nodes are functions, edges are direct call sites.
  - Function body availability info
    - Not available: will be linked in from other unit
    - Overwrittale: Can change during linking (as for PIC libraries)
    - Available
    - Local: Body is known and is called only directly within unit.
  - API for cloning, removing and late construction of functions.

# Basic interface to IPA pass

IPA interface

- Callgraph
  - Nodes are functions, edges are direct call sites.
  - Function body availability info
    - Not available: will be linked in from other unit
    - Overwrittale: Can change during linking (as for PIC libraries)
    - Available
    - Local: Body is known and is called only directly within unit.
  - API for cloning, removing and late construction of functions.
- Varpool holing static variables
  - API mostly symmetric to callgraph.

# Planned IPA passes

- Function call regularization (me or Matrin Jambor)
  - Removal of unused arguments
  - Scalar replacement
  - Pass by value rather than by reference

# Planned IPA passes

- Function call regularization (me or Matrin Jambor)
  - Removal of unused arguments
  - Scalar replacement
  - Pass by value rather than by reference
- Constant propagatation
  - Construct multiple function clones when needed
  - Work in OO environment (ie propagate on aggregates) (Martin Jambor)

# Planned IPA passes

- Function call regularization (me or Matrin Jambor)
  - Removal of unused arguments
  - Scalar replacement
  - Pass by value rather than by reference
- Constant propagatation
  - Construct multiple function clones when needed
  - Work in OO environment (ie propagate on aggregates) (Martin Jambor)
- Matrix flattening (Razya Ladelsky)
- IPA points-to analysis (Daniel Berlin)
- Structure reorg branch
- Devirtualization

- Memory consumption problems
  - We need to hold whole program in memory
  - GIMPLE representation is bloated
  - Garbage collector does not scale very well (while in swap, marking pass is very slow)

- Memory consumption problems
  - We need to hold whole program in memory
  - GIMPLE representation is bloated
  - Garbage collector does not scale very well (while in swap, marking pass is very slow)
- Compile time problems
  - unit-at-a-time/linktime makes compilation slightly **faster**
  - With inlining we see acceptable slowdown

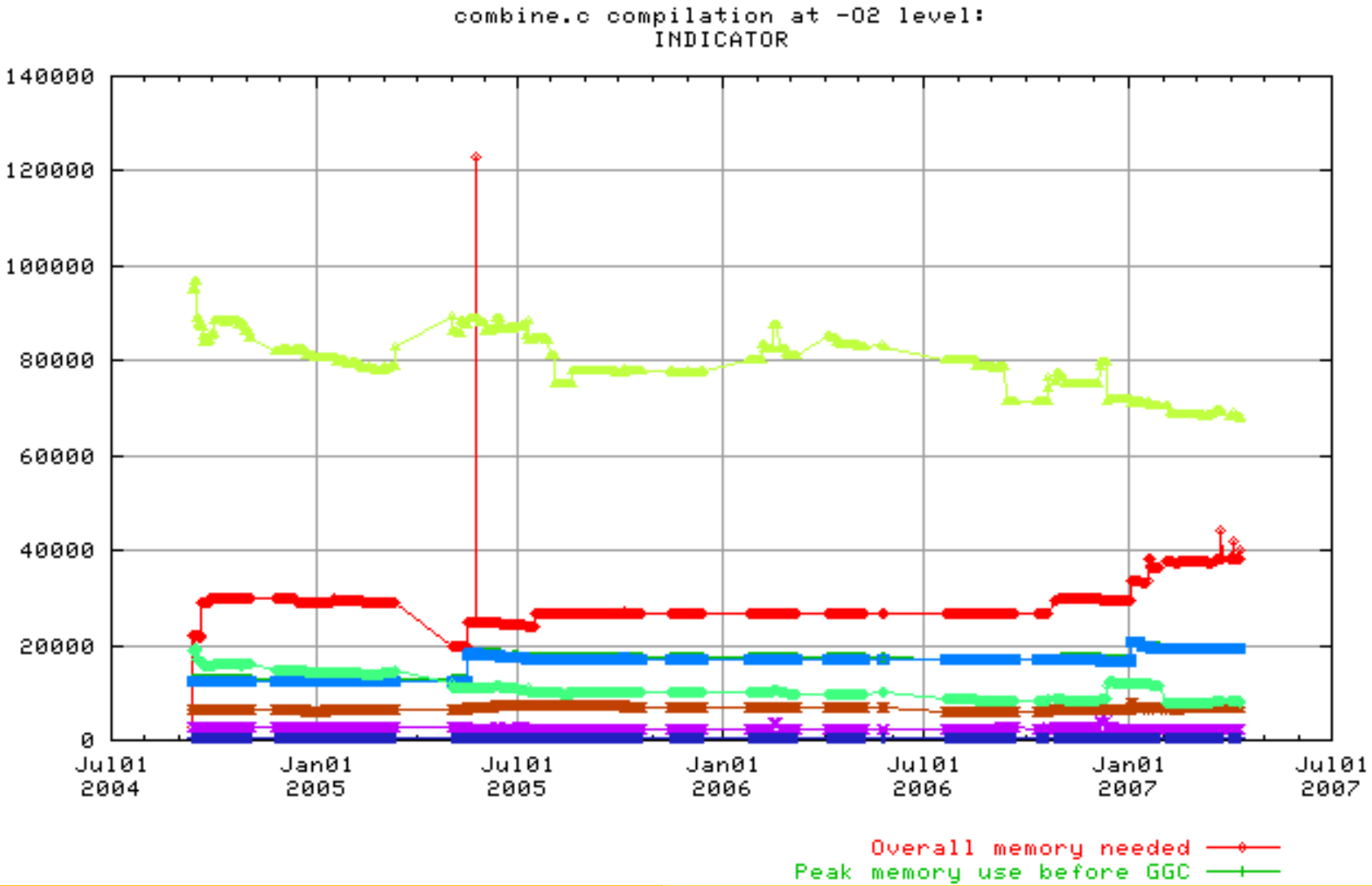
- Memory consumption problems
  - We need to hold whole program in memory
  - GIMPLE representation is bloated
  - Garbage collector does not scale very well (while in swap, marking pass is very slow)
- Compile time problems
  - unit-at-a-time/linktime makes compilation slightly **faster**
  - With inlining we see acceptable slowdown
    - Function-at-a-time: 470s
    - Unit-at-a-time: 512s
    - Intermodule: 572s
    - Whole program: 543s

- Memory consumption problems
  - We need to hold whole program in memory
  - GIMPLE representation is bloated
  - Garbage collector does not scale very well (while in swap, marking pass is very slow)
- Compile time problems
  - unit-at-a-time/linktime makes compilation slightly **faster**
  - With inlining we see acceptable slowdown
    - Function-at-a-time: 470s
    - Unit-at-a-time: 512s
    - Intermodule: 572s
    - Whole program: 543s
  - **however** one can't parallelize linking via **make -j**

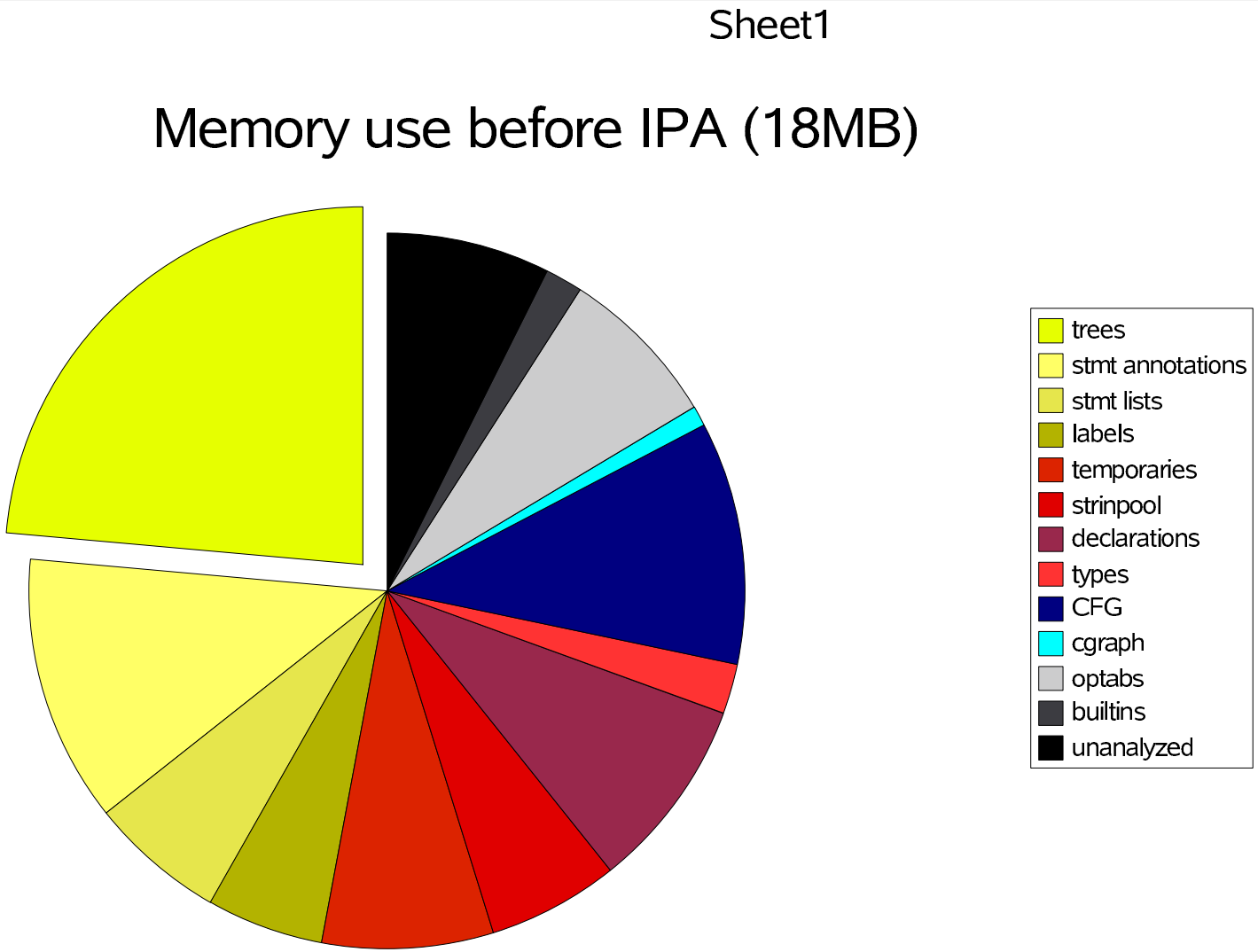
# Memory usage testing infrastructure

- `-enable-gather-detailed-mem-stats` mode
  - Per function call statistic of allocated/leaked GGC objects
  - Bitmap and allocpool statistics
  - Per tree-node statistics of allocated trees
  - Statistics can be gathered after parsing, after IPA and at the end.
- Periodic memory tester at `http://www.suse.de/~ gcctest/memory`
  - Daily testing of `-enable-gather-detailed-mem-stats` on selected testcases.
  - Graphs plotted
  - Archive detailed dumps, so one don't has to rebuild
  - Complains via email with suspected changelog entries.

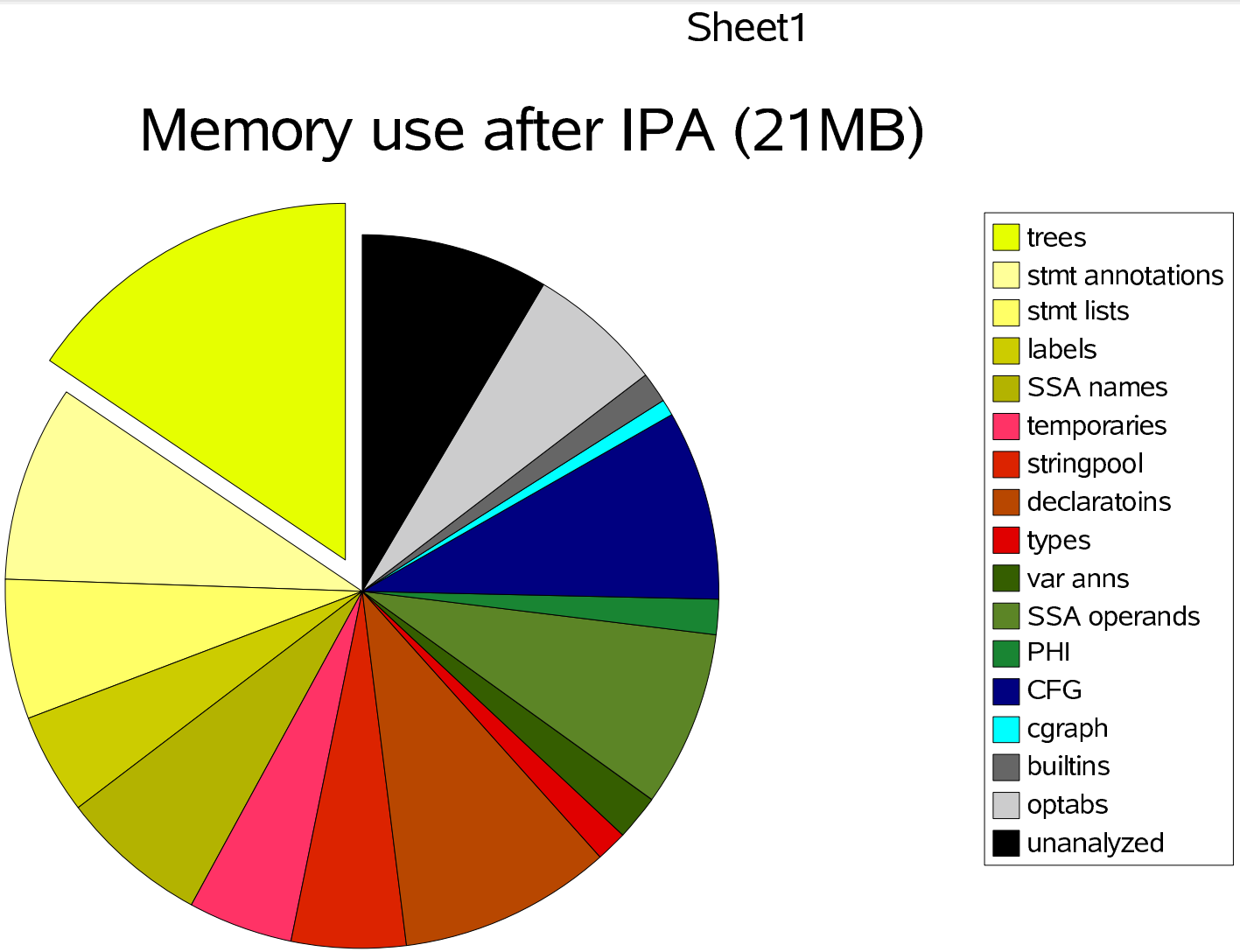
# Memory usage (combine.c)



# Memory usage chart for combine.c



# Memory usage chart for combine.c



# Some memory reduction plans

- Gimple flattening:
  - Stmt annotation (96 bytes)
  - Statement list node (48 bytes)
  - gimple modify stmt (48 bytes)
  - expression (64 bytes)
  - Flattened: next,prev pts (16), basic block (8), operands (48), uid (4), type (8), locator 2$\times$ (8),source/dest (32), operation+flags (8),
  - 256 bytes $\rightarrow$ 132 bytes, 20565 stmts in combine.c, savings of 2.5MB (out of 21MB), 11%

# Some memory reduction plans

- Removal of labels and use of direct BB pointers:
  - Stmt annotation (96 bytes)
  - Statement list node (48 bytes)
  - label decl (208 bytes)
  - label expr (80 bytes)
  - 6237 artifical labels, 4799 live, 2MB savings, 10%
- GIMPLE conditional goto statement?
  - Save 2 goto expression nodes, savings are difficult to estimate precisely.

# Proper lifetimes for datastructures

- Variable and statement annotations are 10%
  - Most of data are local to one or two pass. Don't need to live all at once.
  - In statement annotation only basic block and tree operands are really needed
  - Variable annotations can probably be completely replaced by on-side arrays

# Dangling pointers

Dangling pointer problems

- GCC datastructures have a lot of dead pointers.
- Garbage collector can’t collect datastructures that will not be used again.

# Dangling pointers

Dangling pointer problems

- GCC datastructures have a lot of dead pointers.
- Garbage collector can't collect datastructures that will not be used again.
- Samples:
  - Unused field in variable annotation $\rightarrow$ dead statement $\rightarrow$ list of dead SSA nodes $\rightarrow$ dead statements in function.
  - Inliner makes pointers to abstract origina $\rightarrow$ dead function bodies $\rightarrow$ dead template bases $\rightarrow$ dead C++ frontend data. Abstract origins are unused for anything but decls.

# Dangling pointers

Dangling pointer problems

- GCC datastructures have a lot of dead pointers.
- Garbage collector can't collect datastructures that will not be used again.
- Samples:
  - Unused field in variable annotation → dead statement → list of dead SSA nodes → dead statements in function.
  - Inliner makes pointers to abstract origina → dead function bodies → dead template bases → dead C++ frontend data. Abstract origins are unused for anything but decls.
- Explicit `ggc_free` is useful to identify dangling pointers. (with checking, GGC ICE when sees dangling pointer)

# Dangling pointers

Dangling pointer problems

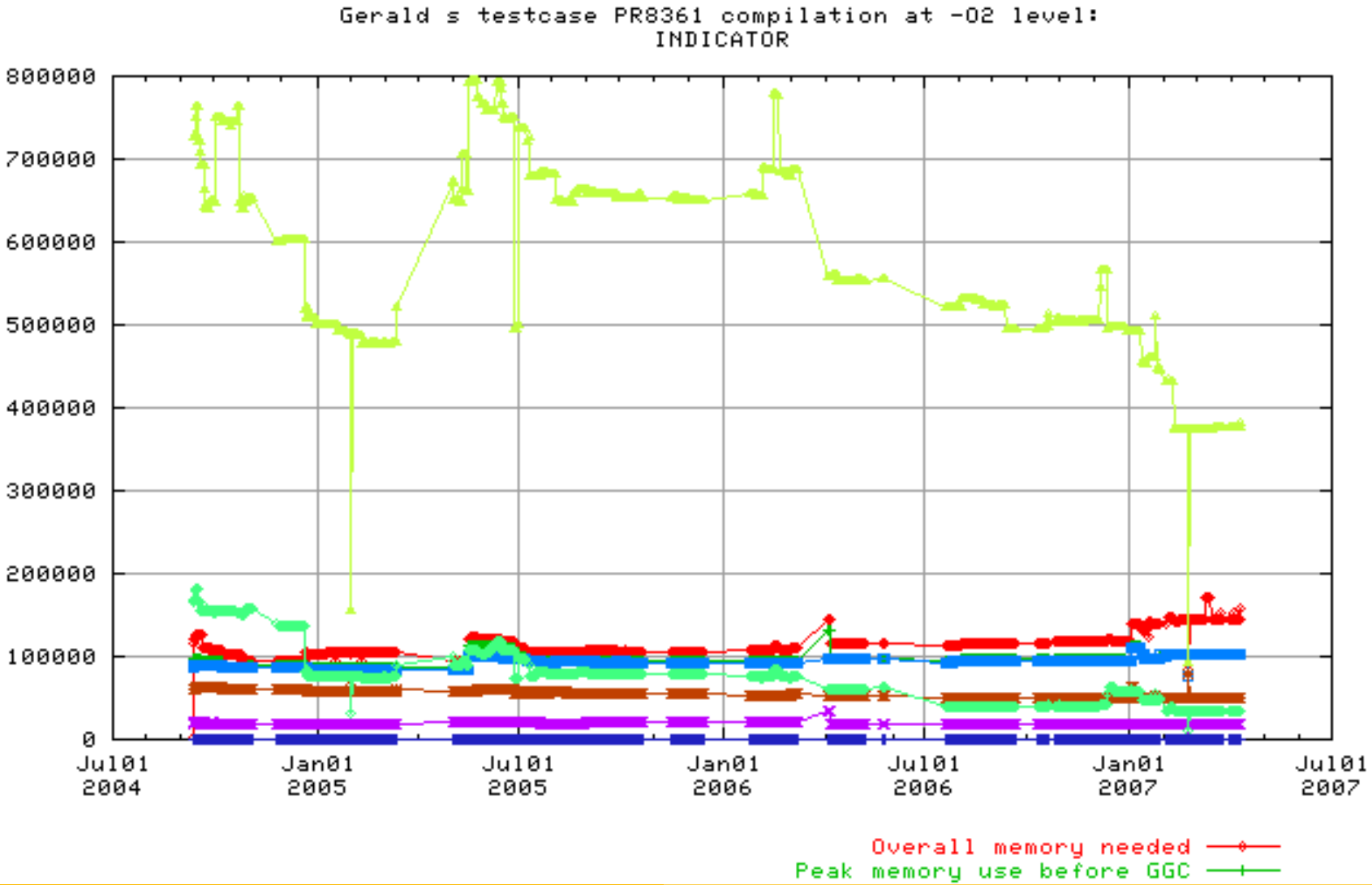
- GCC datastructures have a lot of dead pointers.
- Garbage collector can't collect datastructures that will not be used again.
- Samples:
  - Unused field in variable annotation → dead statement → list of dead SSA nodes → dead statements in function.
  - Inliner makes pointers to abstract origina → dead function bodies → dead template bases → dead C++ frontend data. Abstract origins are unused for anything but decls.
- Explicit `ggc_free` is useful to identify dangling pointers. (with checking, GGC ICE when sees dangling pointer)
- `ggc_free` use drives people mad.

# Dangling pointers

Dangling pointer problems

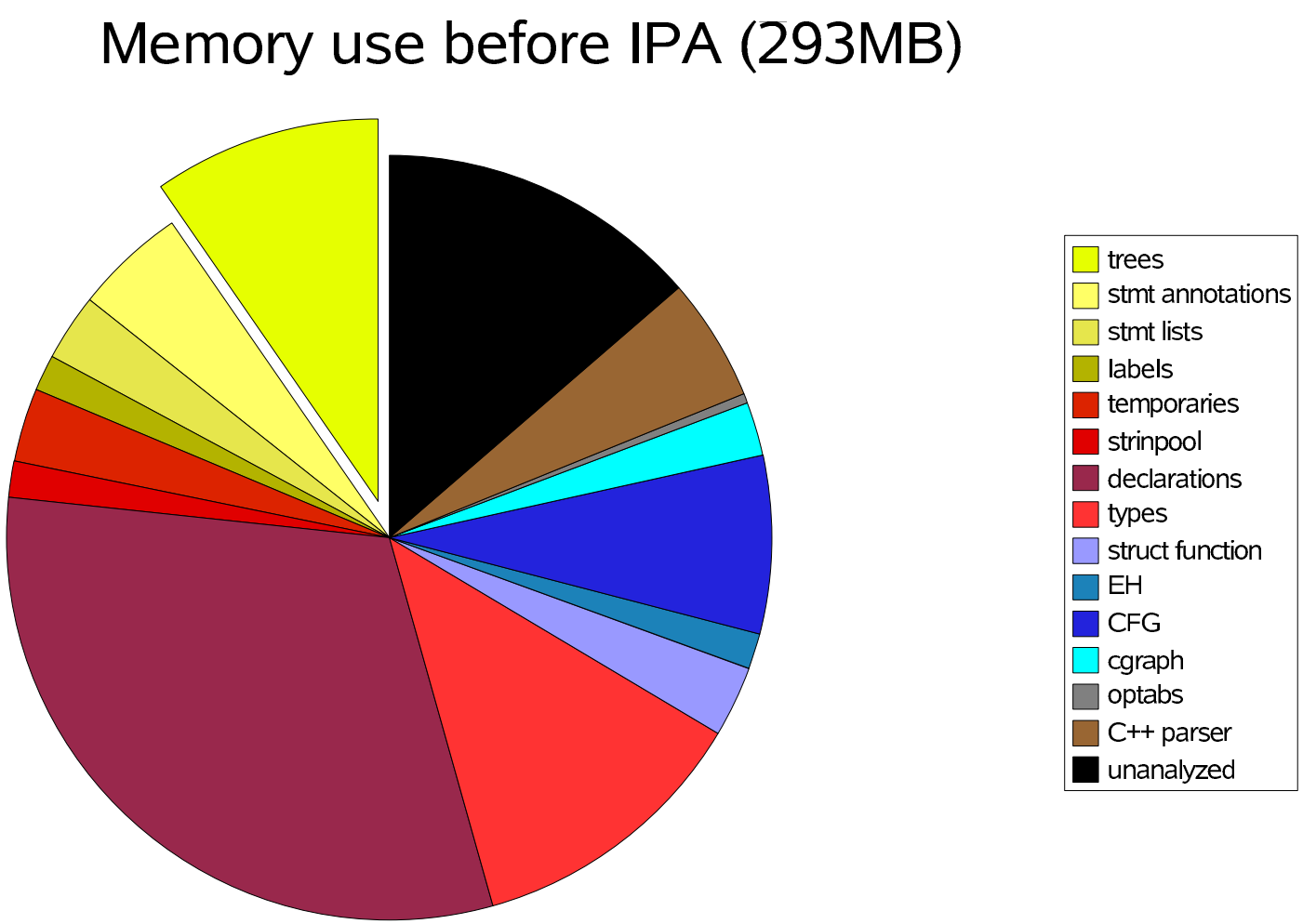
- GCC datastructures have a lot of dead pointers.
- Garbage collector can't collect datastructures that will not be used again.
- Samples:
  - Unused field in variable annotation → dead statement → list of dead SSA nodes → dead statements in function.
  - Inliner makes pointers to abstract origina → dead function bodies → dead template bases → dead C++ frontend data. Abstract origins are unused for anything but decls.
- Explicit `ggc_free` is useful to identify dangling pointers. (with checking, GGC ICE when sees dangling pointer)
- `ggc_free` use drives people mad.
- Plan: `ggc_unreferenced` function that translate to noting with release compiler, behaves like `ggc_gree` otherwise.

# Memory usage (PR8361.cc)



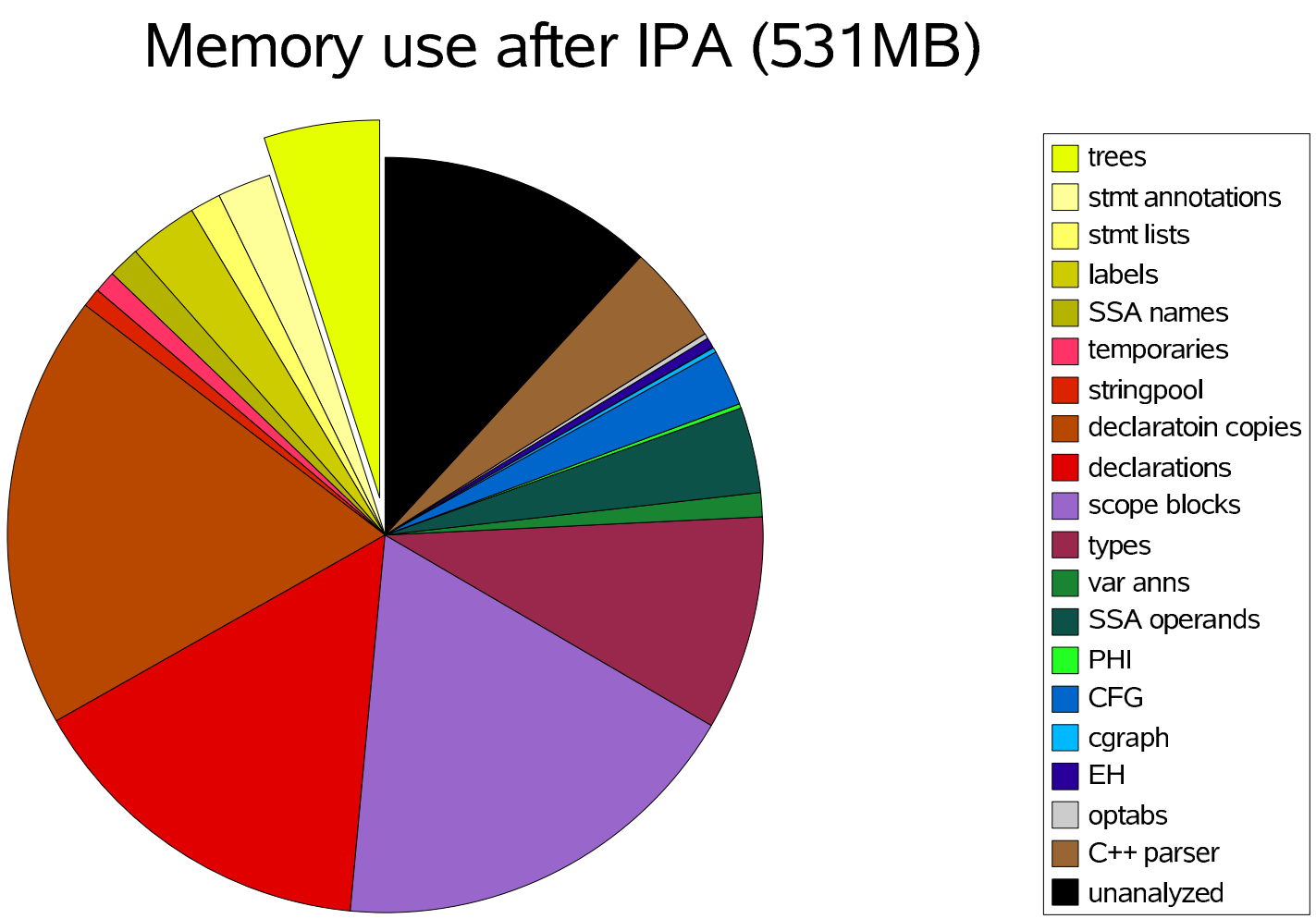
# Memory usage (tramp3d.cc)

Sheet1



# Memory usage (tramp3d.cc)

Sheet1



# Compilation time issues

- Obviously we should get GCC faster
- Link time allows parallelizing the parsing part
- We should push as much work to parsing section as possible
  - SSA conversion
  - Local optimization
  - Conservative local interprocedural optimization (inlining) targetted to reduce code size.
  - On-disk representation should be close enough to GIMPLE to make conversion fast and efficient.
- GCC can be parallelized: after IPA, the function bodies can be sent to separate compilation processes.
- GCC can be threaded but it is hard. Perhaps makes sense for IPA passes.

# Benchmarks running

- Spec2000 seems to be rather poor for basic IPA
  - Improvmeents possible in number of benchmarks (VPR, Art etc.). All are rather special case transformations of poor datastructures.
  - Struct-reorg, mattrix-flattening and other projects implement it for GCC. Not included in my benchmarks.
- SUSE has IA-64 SPEC2000 tester. It overheated last summer, but it is to be re-started soonish.

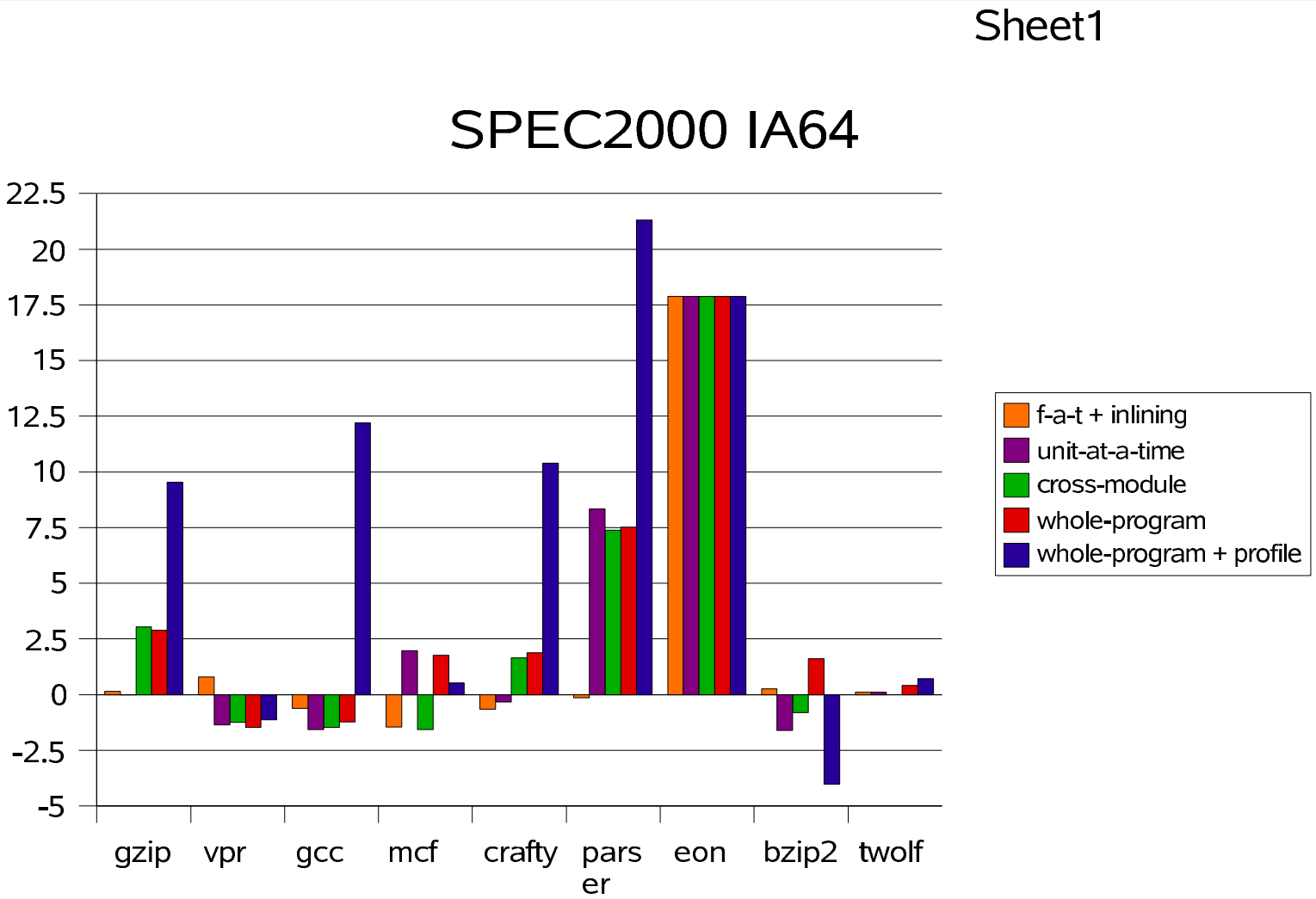
# Benchmarks running

- Spec2000 seems to be rather poor for basic IPA
  - Improvmeents possible in number of benchmarks (VPR, Art etc.). All are rather special case transformations of poor datastructures.
  - Struct-reorg, mattrix-flattening and other projects implement it for GCC. Not included in my benchmarks.
- SUSE has IA-64 SPEC2000 tester. It overheated last summer, but it is to be re-started soonish.
- Spec2006 results are not publically available.

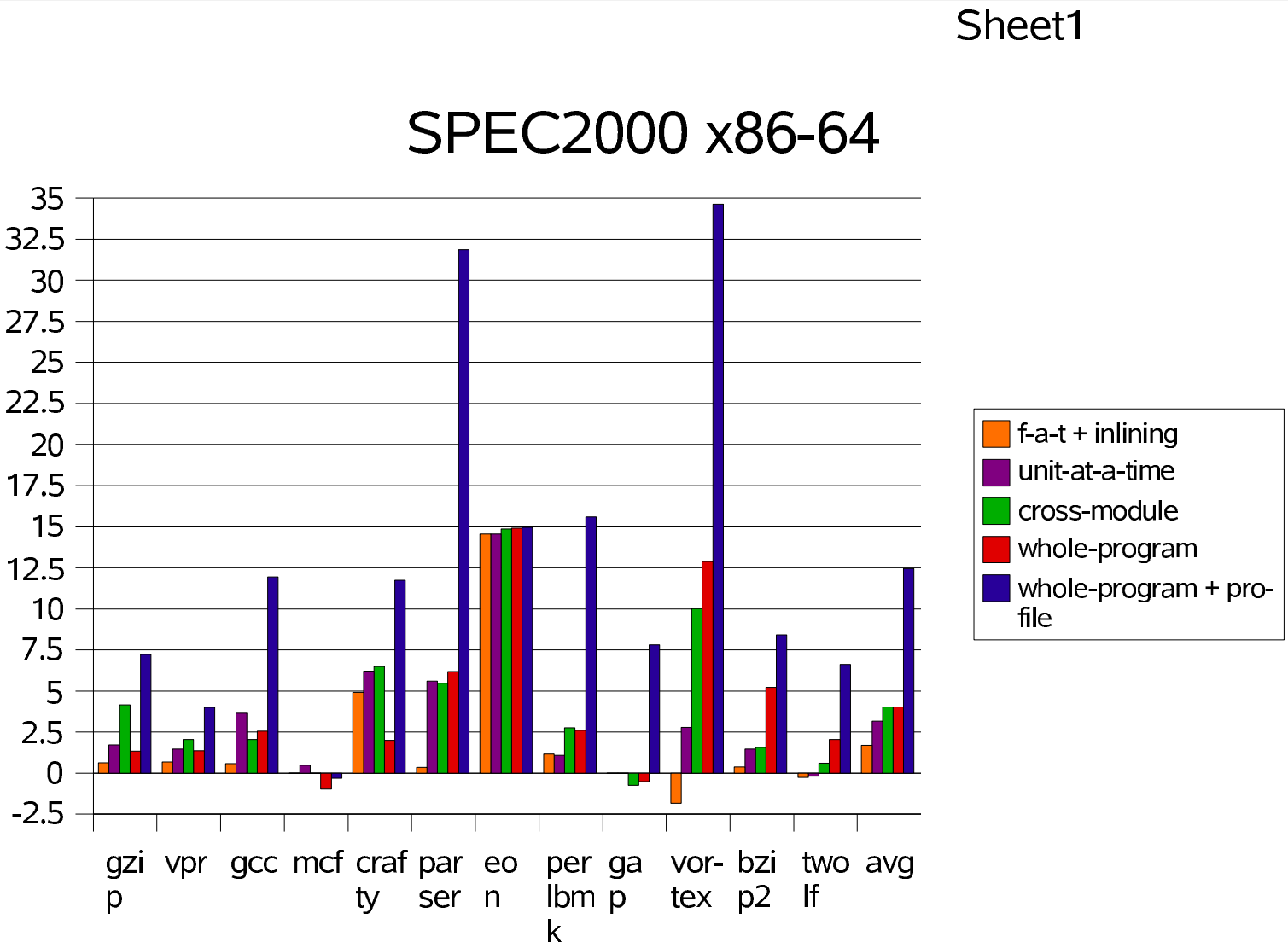
# Benchmarks running

- Spec2000 seems to be rather poor for basic IPA
  - Improvmeents possible in number of benchmarks (VPR, Art etc.). All are rather special case transformations of poor datastructures.
  - Struct-reorg, mattrix-flattening and other projects implement it for GCC. Not included in my benchmarks.
- SUSE has IA-64 SPEC2000 tester. It overheated last summer, but it is to be re-started soonish.
- Spec2006 results are not publically available.
- We put together C++ centric benchmark suite. It is testing programs with high abstraction penalty.
  - **Tramp3d** (hydrodynamics simulation)
  - **Wave** (C++ preprocessor written in spirit)
  - **Botan** (Cryptographic library)
  - **Freefem3d** (finite element library)
  - **DLV** (Disjunctive Datalog System)

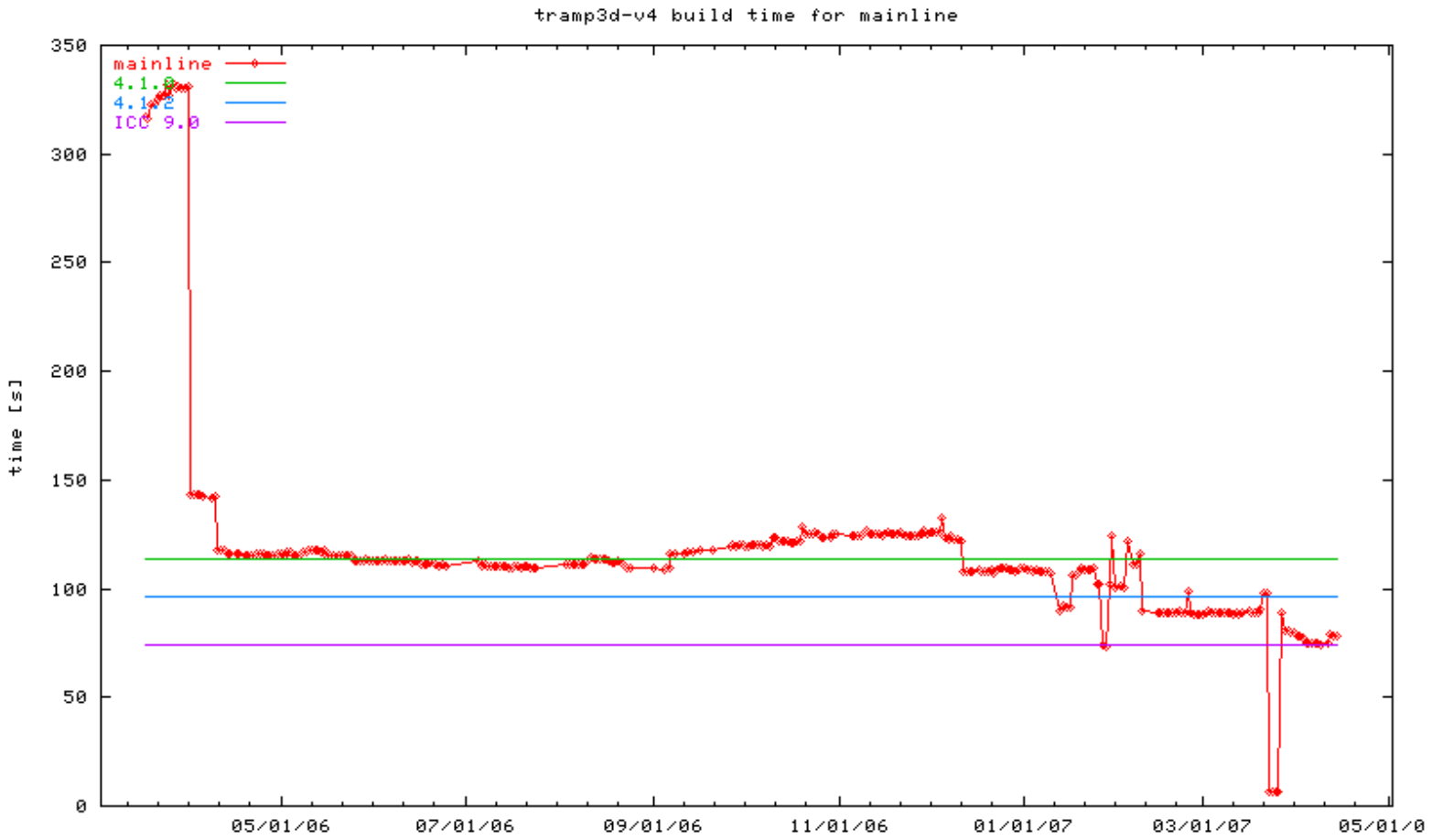
# Partial SPEC2000 results (ia64)



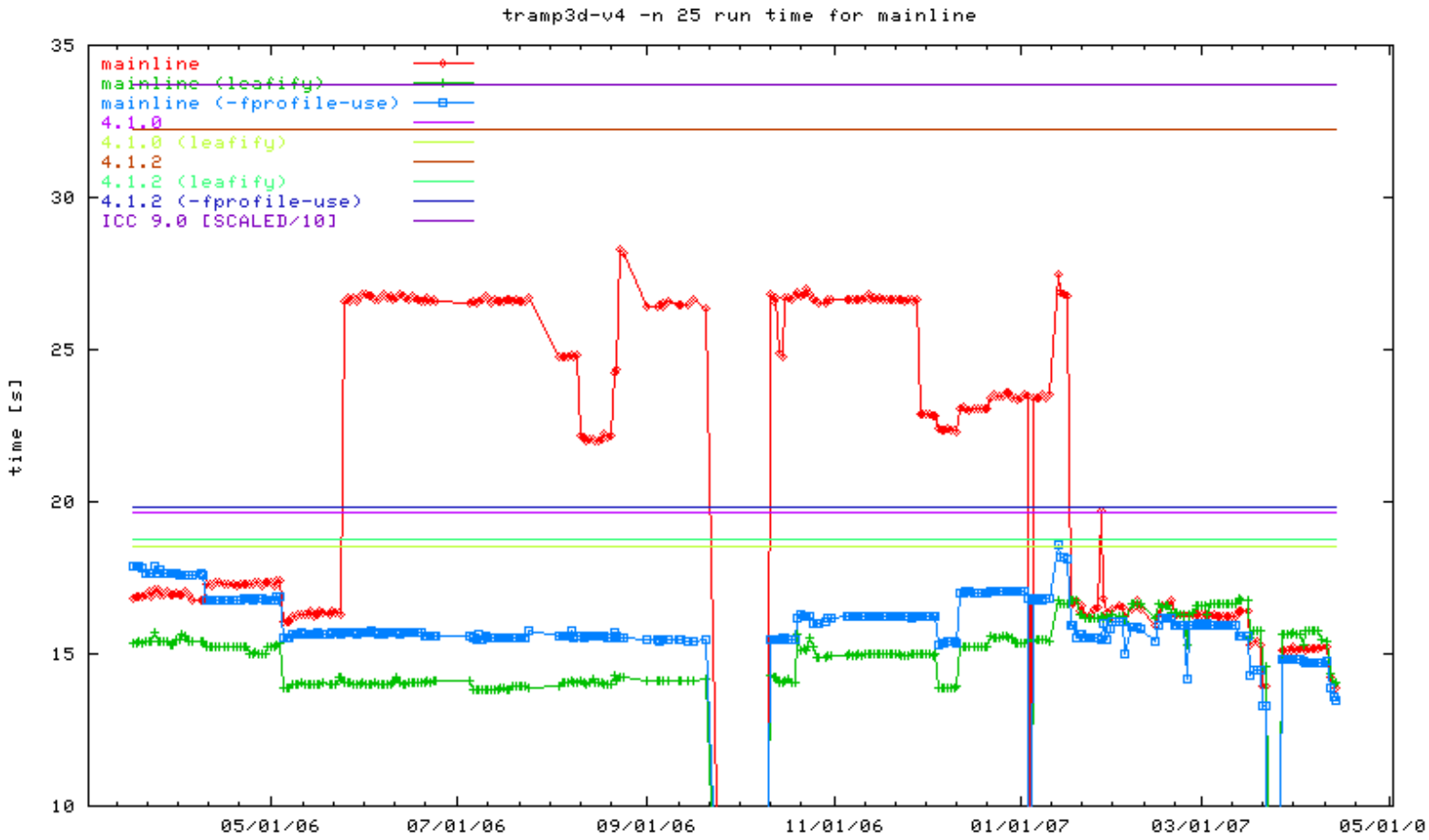
# Partial SPEC2000 results (x86-64)



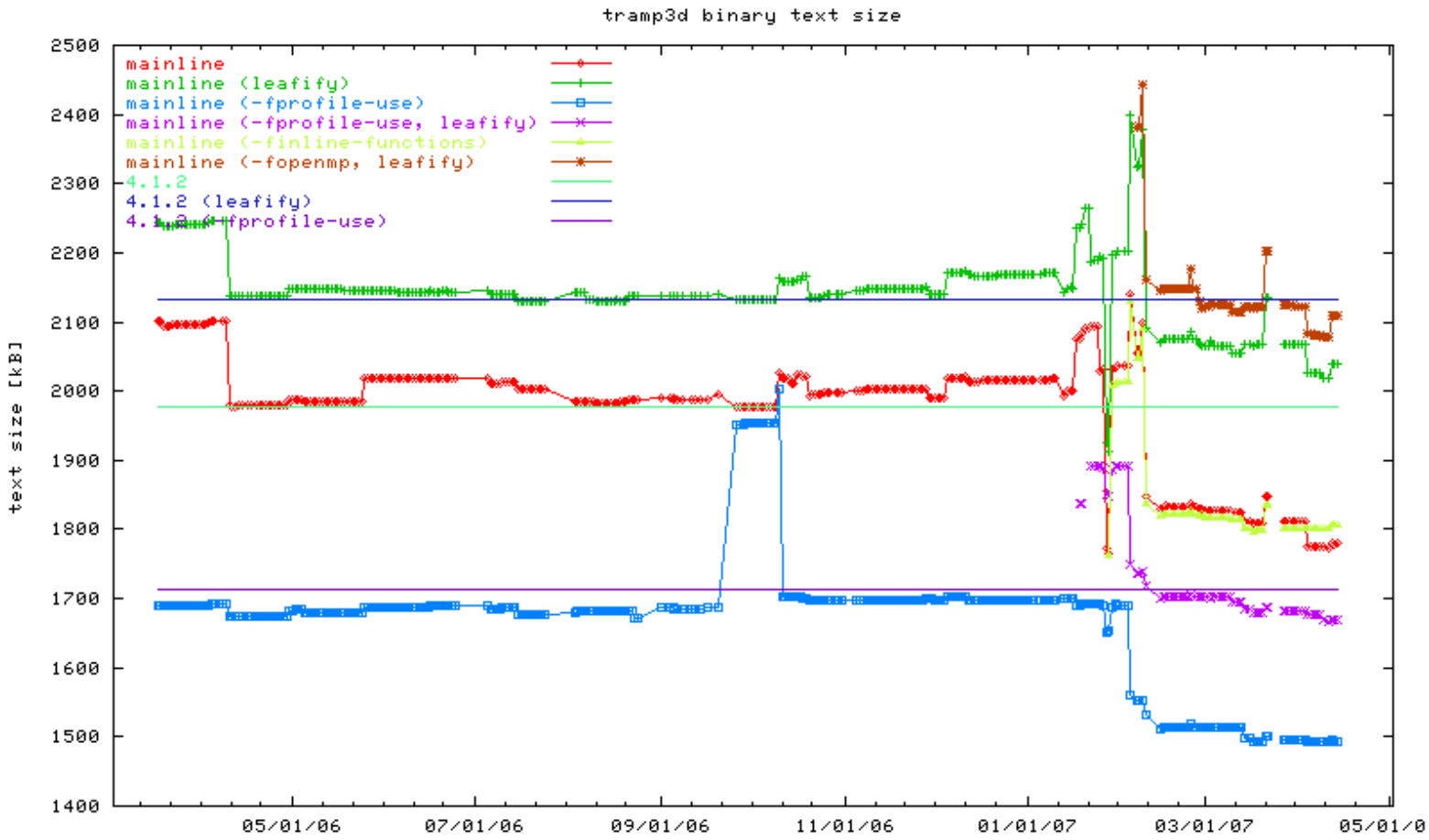
# Tramp3d build time (x86-64)



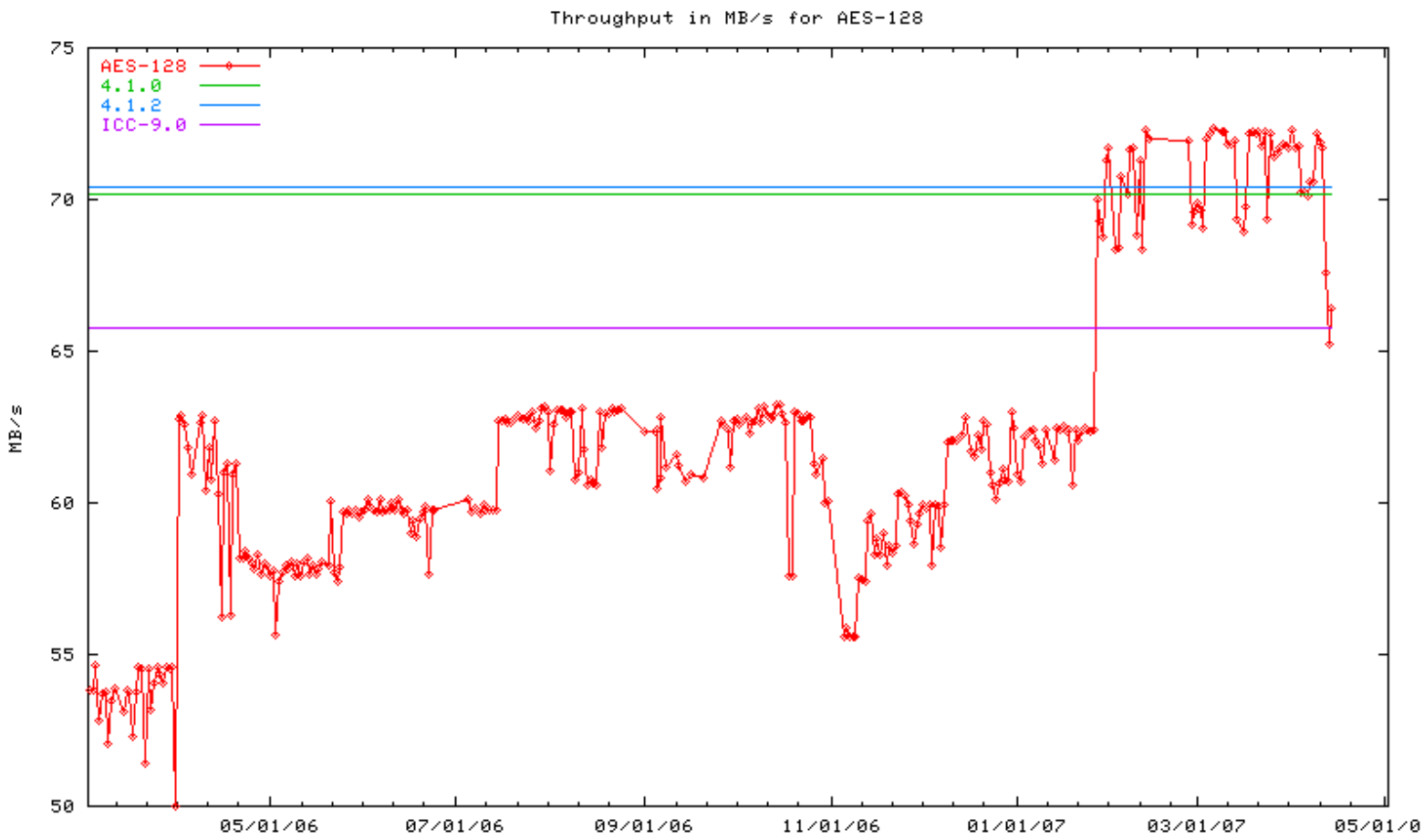
# Tramp3d runtime



# Tramp3d file size



# Botan AES-128 troughput



# The missing benchmarks

- Larger C applications with whole program IMA
  - SPEC2000 still don't compile by default with IMA due to type problems
  - We might be able to find better benchmark application(s)
- Larger C++/fortran applications with whole program IP
  - For C++/fortran we usually can just concatenate the units
  - Fortran needs to be made to work with cgraph
- Java LTO
  - Java from bytecode compilation is LTO
  - We might pick good CPU bound Java applications to excercise future LTO memoru usage/code quality issues

# Thank you!

Questions?